IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 7, 2020, accepted March 25, 2020, date of publication April 24, 2020, date of current version May 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2990331

Packet Classification Using GPU and One-Level Entropy-Based Hashing

SHLOMO GREENBERG, (Member, IEEE), TOMER SHEPS, DAVID A. LEON, AND YEHUDA BEN-SHIMOL, (Member, IEEE)
School of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Beer Sheva 84105, Israel.
Corresponding author: Shlomo Greenberg (shlomog@bgu.ac.il)

This work was supported in part by the HiPer Consortium within the frame of the Israeli Innovation Authority's MAGNET Program.

ABSTRACT The demand for on-line analyzing of internet traffic for both security and QoS consideration directly increases as a function of using diverse applications and as malicious attacks increase. This paper presents a new fast parallel packet classification algorithm based on entropy hashing. The algorithm uses a one-level hashing data structure and enables partitioning a very large rules-set into smaller uniformly distributed sub-rules look-up tables based on maximum entropy and Most Significant Bit (MSB) pattern hash keys. This minimizes the classifier searches only to the relevant appropriate look-up table using the same hash key, and therefore significantly shortens the classification process. A further speed-up factor is achieved by parallelizing the classification algorithm using Nvidia Graphics Processing Unit (GPU). The proposed algorithm is applied to both ACL and FW applications using common synthetic rules-sets of size up to 500k rules. The simulation results show that the proposed algorithm outperforms existing classifiers in terms of both speed up and memory utilization. The required memory size is significantly reduced, and a classification speed-up factor of up to 200 is demonstrated compared to a similar serial approach.

INDEX TERMS Packet classification, hashing, entropy, information gain, GPU, parallelism.

I. INTRODUCTION

The rapid increase of both network bandwidth and malicious attacks, as well as the increased use of diverse applications, requires analyzing and controlling internet traffic for both security and QoS (Quality of Service) considerations while gaining line speeds [1]. Packet classification is known as the process of selectively identifying different classes of internet packets, classifying them into flows, and applying the same treatment to all packets belonging to the same flow according to certain predefined rules. To classify a packet as belonging to a specific flow or set of flows, a packet classifier must perform a fast search over a large set of filters and be able to find the matching rule among all the rules in a large rules-sets using multiple fields of the packet header [2].

To face the growing need of real-time on-line packet classification, while supporting wire speed of hundreds of Gbps, various classification algorithms have been developed, based on various data structures (e.g., Tries, Hash tables, etc.) [3]–[6], and various platforms such as pure CPU software, ASIC, FPGA and GPUs [7]–[11]. Some architectural solutions for packet classification suggest the use of Ternary Content Addressable Memory (TCAM) devices [10], [12]–[14] in order to meet performance constraints imposed by high-speed links [2]. However, TCAMs suffer from high cost, high power consumption, storage inefficiency, and limited scalability to long input keys [2]. Taylor [2] provides a survey and taxonomy of packet classification techniques describing some high-level classification approaches. Some naive approaches use a linear search to examine all entries in the rules-set (which are pre-arranged in decreasing priority order) against the incoming packet. This may lead to an exhaustive search in case of large rules-sets. Therefore, most approaches suggest using a pre-processing phase for partitioning the original rules-set into smaller sub-tables using decision tree [6], [15], or hashing function [16], [17].

A common approach makes use of a decision tree data structure [4], [6] to graphically represent the rules-set. The decision tree is constructed from all the rules in a given rules-set, in such a way that each level in the tree represent a different header field (of the incoming packet), and each leaf represents a specific rule (or sub-sets of rules). Then, in the classification stage, the algorithm uses the packet header

The associate editor coordinating the review of this manuscript and approving it for publication was Yulei Wu.

fields to traverse the decision tree in a top-down manner [6], [15], [18]. Dong *et al.* [15] propose a heuristic method, based on information entropy, to efficiently build a classification decision tree with relatively small storage requirements. The decomposition approach suggests splitting the multiple fields search into instances of single field searches, performing an independent search on each packet field, and then combines the results [8], [19]. However, most algorithms that support multiple fields have been designated for limited rules-set sizes of up to 15k rules [2].

Gupta and McKeown [20] present an iterative *Recursive Flow Classification algorithm*, for which in each iteration only a subset of the inspected fields is used to generate a cross-product table, and all unmatched entries are eliminated for storage saving. Rules-set partitioning is usually carried out using some sorting criteria (such as entropy or MSB) to define different flow categories [15], [16], [21]. For example, a *tuple space* approach partitions the rules-set according to some specified bits of each field in the filters, and then probe a sub-table using simple exact match searches [22]. Two other well-known tuple-based algorithms, *rectangle search* [23] and *Entry Pruned Tuple Search* [24], have been proposed to improve the performance of the tuple space search.

When adopting an existing packet classification approach, a tradeoff between speed performance and high storage efficiency should be considered. Algorithms that present superior speed performance usually suffer from the problem of high memory requirements [25]. Although, the algorithms based on decision trees [26]–[28] and cross-product [20], [29] present the fastest search performance they suffer from poor storage efficiency, whereas the *Pruned Tuple Space Search* [24] and *Independent Sets* [30] algorithms typically have the least storage requirements [21]. Hasan *et al.* [17] describe three major techniques families for performing *Longest Prefix Matching* (LPM): using *Ternary Content Addressable Memory* (TCAM), tries-based schemes, and hash-based schemes. A collision-free hashing scheme proposes an efficient storage approach and provides both fast classification and significantly lower power consumption compared to TCAM [31]. Moreover, unlike tries, hash tables employ a flat data-structure, achieving potentially small memory sizes suitable for on-chip storage, and key-length-independent $O(1)$ latencies [17].

Hardware-based algorithms are used to accelerate existing classification algorithms using different hardware platforms such as FPGAs [7], [10], [11], [32], multiple-core CPUs [8], and GPUs [22]. Some packet classification algorithms have been implemented using GPU to maintain line speed [22], [33], [34]. Varvello *et al.* [22] demonstrate GPU-accelerated versions for three classification search algorithms: linear search, tuple search, and bloom filters search. The bloom filters algorithm demonstrates the highest classification speed (up to 115 Mpps) for small rules-sets (up to 5k rules). However, the performance dramatically decreases (down to 20 Mpps) for large rules-sets of above 100k rules. Yang *et al.* [35] suggest a *Discrete Bit Selection* (DBS) algorithm, for which some effective bits are consecutively chosen from the packet header fields and serve as the hash key mask in filling and probing the hash table.

A. RELATED WORK

Kang and Deng [33] propose a GPU-based linear search framework using a meta-programming technique for packet classification. They investigate the previous DBS hash-based algorithm and demonstrate a speedup factor of 17 in comparison to a CPU-based implementation. The GPU implementation delivers an average throughput of 10.7 and 4.8 Mpps with rule number of 500 and 2k, respectively. Zhou *et al.* [34] propose a GPU implementation of a range-tree search and a decomposition-based bit vector (BV) tree approach. The algorithm scales well across a range of rules-set sizes from 512 to 4k rules and demonstrates a throughput improvement factor of 1.9 compared with the implementation of another multi-core platform.

Choi *et al.* [16] use a heuristics approach for choosing the specific packet header fields used for hashing. A combination of 17-bits from the port and protocol fields, with 16-bits selected from the IP source and destination addresses, were examined. They propose two-level hashing using two different hash keys: a first level hashing based on maximum entropy derived from the port and protocol fields, and a second level hashing based on the MSB pattern of the IP fields. Although this approach presents promising results in terms of speed performance (207 memory accesses for 500k rules), it suffers from a very poor memory utilization and requires a huge storage memory (32 GB in the worst-case scenario). This disadvantage is not acceptable for on-chip and GPU hardware implementation. Motivated to improve the memory consumption and still utilize the potential speed up by using hashing, we propose a new one-level unique hashing algorithm based on information entropy and a GPU-based implementation for achieving further significant speed-up factor using parallelism approach.

Varvello *et al.* [22] analyzed typical non-cryptographic universal hash functions due to their computation simplicity and implemented equally-sized sub-tables using cuckoo hashing [22], [36] , which provides constant lookup time. However, this method may require rehashing, that is, selecting a new set of hash functions and reinserting all rules.

This paper presents a parallel hash-based packet classification algorithm. The proposed algorithm uses a one-level hashing data structure and enables partitioning a very large rule-base list into smaller uniformly distributed sub-rules lookup tables based on maximum entropy and MSB pattern hash keys. A further speed-up factor is achieved by parallelizing the classification algorithm using NVIDIA GTX 1080 GPU. The proposed algorithm is applied to both ACL and FW applications, using common rules-sets of size up to 500k rules. The simulation results show that the proposed algorithm outperforms existing classifiers in terms of both speed up and memory utilization. The required memory size is significantly reduced, and a classification speedup factor of

more than 200 is demonstrated compared to a similar serial approach.

The rest of this paper is organized as follows: Section II presents the proposed classification algorithm and the implementation approach. Section III describes the GPU-based parallel implementation. Section IV presents the experiments and results. Finally, conclusions and summary are given in Section V.

II. THE PROPOSED ALGORITHM AND IMPLEMENTATION METHODOLOGY

This section formalizes the packet classification problem and presents the proposed classification algorithm in detail. The main idea is to efficiently partition a large rules-set into smaller sub-tables enabling fast line speed classification while using minimal memory resources. Different kind of hash functions is proposed to uniformly distribute the original rules into a hash table composed of 256k entries for 18 bits hash key. Each entry in the hash table contains a 64-bits pointer to a different small sub-table and is associated with a specific dynamic rule array with a typical size of less than 1k rules. Therefore, only one memory access is required for approaching a specific rule array, for which a linear search is performed to find the final rule match. The classification process is composed of two main phases: the off-line pre-processing phase and the real-time packet matching phase. In this section, we first present the problem definition and describe the rules structure and its efficient representation. Then, the proposed hash functions and the selected hash keys are presented, and finally, the two phases of the proposed classification algorithm are described.

A. PROBLEM DEFINITION

The problem of packet classification is defined as the process of matching an incoming packet to one or more predefined rules from a set of n rules according to given criteria. Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of n rules. Each rule r_i is described as $r_i = \{c_1, c_2, \ldots, c_m, priority, action\}$ and composed of m criteria and two attributes: *priority* and *action*. Each criterion represents a match condition such as an exact match, prefix match or range checking. Each rule is examined for a match with some specific fields in the packet header according to different criteria.

A *priority* attribute is assigned to each rule to handle the case for which an incoming packet matches multiple rules. In the case of a match, a specific action is applied to the incoming packet, according to the action attribute of the matched rule. Typical rules-sets are application-oriented and are differently characterized using different header fields. In this work, we use two common rules specifications: Access Control List (ACL) and Firewall (FW). ACL specifications are characterized mainly by filtering IP-addresses and therefore have more specific IP fields, while FW is more related to applications and protocol filtering and therefore the rules list contains more specific port and protocol fields [37].

TABLE 1. 5-tuple rule-set example*.

Rule	SA	DA	SP	DP	Protocol	Priority	Action
r_1	*	*	2-9	6-11	*	1	A_0
r_2	1*	0*	3-8	1-4	10	2	A_0
r_3	0*	0110*	9-12	10-13	11	3	A_1
r_4	0*	11*	11-14	4-8	*	4	A_2
r_5	011*	10*	1-4	9-15	10	5	A_2
r_6	011*	11*	1-4	4-15	10	5	A_1
r_7	110*	00*	0-15	5-6	11	6	A_3
r_8	110*	0110*	0-15	5-6	*	6	A_0
r_9	111*	0110*	0-15	7-9	11	7	A_2
r_{10}	111*	00*	0-15	4-9	*	7	A_1

* Derived from [33]

A packet p is represented by an ordered m-tuple $\{f_1, f_2, \ldots, f_m\}$. The elements of the m-tuple are extracted from specific fields of the packet p, for example, source and destination IP addresses. The m-tuple complies to a predefined rule $r = \{c_1, c_2, \ldots, c_m, priority, action\}$ if each of the elements of the m-tuple matches each of the m criteria correspondingly.

Although this work addresses the general problem of packet classification, the common IPv4 protocol, and the specific 5-tuple classification, are used as an example for evaluation purposes and comparison with previous work. However, the proposed approach can be easily applied to IPV6 as well, and work well with any different kind of general n-tuples. The appropriate selection of the fields used for the hash key is critical for proper flow classification.

The classification of IPv4 packets usually uses a 5-tuple classification using 5 fields of the packet header: IP source address (SA) and IP destination addresses (DA), each 32 bits in length, source port (SP) and destination port (DP) numbers, each 16 bits in length and a transport-layer protocol (8-bits). Table 1 shows a simplified example taken from [32], where each rule contains different match conditions for the 5 header fields: prefix match for SA/DA, range match for SP/DP, and an exact match for the protocol field. A packet is being considered to successfully match a rule only if all the fields within that rule are matched. In the case of a match, the action attribute of the matched rule is carried out. In case an incoming packet matches multiple rules, only the highest prioritized rule is considered and its specific action is the one to be applied to the packet.

B. RULES DATABASES AND EFFICIENT REPRESENTATION

Each rule can be related to several header fields with different kind of matching types (like exact and long prefix match) using wildcards to allow multiple addresses combination and range checking. Our test data is generated using the common *ClassBench suite* of tools [37] to provide synthetic rule-sets representing real-life rules. The *ClassBench* produces synthetic rules-sets that accurately model the characteristics of real rules-sets. The tools provide varying size and complex rules-sets, along with a sequence of packet headers to exercise the synthetic rules-sets. The size of the rule-sets varies from thousands to hundreds of thousands of rules. The rules

databases which have been used to evaluate the classification performance in this work are synthesized from two typical applications: ACL (Access Control List) and Firewall.

The *ClassBench* adapts the Classless Interdomain Routing (CIDR) [38] allowing to specify a subnet within an existing network, e.g. the source IP field may contain the form 192.0.0.0/1. The use of the prefix match enables binding multiple possible exact matches into a single rule. A priority and a specific action are assigned to each rule. Although a 5-tuple representation requires only 13 bytes for classification, a typical rule may contain up to 32 bytes, including IP source and destination addresses as well as Ports ranges, protocol, action type (accept or deny) and a priority field (8-16 bits). The data structure of each rule is zero-padded to get a uniform memory data alignment. Efficient rule representation is achieved by converting the SA and DA fields into ranges using a range matching condition in place of a prefix match. This allows the matching to be carried out by checking a range condition using only two comparison operations rather than using several bit-wise operations (up to 32 XOR operations) per a prefix match.

C. HASH TABLE AND KEY FUNCTIONS

The proposed algorithm uses a one-level hashing data structure and enables partitioning a very large rules-set into many much-smaller, uniformly distributed sub-rules lookup tables, based on *maximum mutual information* and *Most Significant Bit* (MSB) pattern hash keys. Then, the same hash keys, used for partitioning the rules-set, are also used by the classifier. This minimizes the classifier searches to the relevant sub ruleset to which the incoming packet belongs to, and therefore significantly shortens the classification process.

The proposed hash key is constructed from some selected bits (16-18 bits) from the packet header fields which compose the classification space. The number of the chosen bits defines the hash key length, and therefore the size of the hash table, that is, for 16-bits key length a hash table with 64k entries is created. Each entry of the hash table points to a specific smaller sub ruleset. Therefore, the classifier needs only one memory access to approach any of the sub rulesets. Then, a linear search is performed to match the incoming packet with the rules belonging to that sub ruleset. The use of only one hashing level together with a proper selection of the hash key bits, and performing a linear search on a much smaller sub-rules table has been found to be efficient in terms of both speed and memory consumption for rules sets of up to 500k rules. The selection of the hash-key length determines both the hash table size and the number of the rules sub-tables. Although, a choice of shorter hash-key yields a smaller hash table, and fewer sub-tables, the average sub-table size increases. Since the size of a sub-table directly affects the linear search performance, a trade-off between memory size and classification performance should be considered. The selection of 16-bits and 18-bits hash keys results with a reasonable hash-table size of 512k and 2M and a total required memory of 130M and 290M correspondingly (for 100k rules-set). However, the 18- bits key is preferable since the average sub-table size is lower (about half the size of 16-bits for large rule-sets), and therefore the linear search time is twice faster. The choice of a larger hash-key yield a much larger memory size without significant improvement of the linear search.

The preferred hash key type is determined according to the distribution of the given rule dataset. The given rules are distributed into sub-tables according to the selected hash type (i.e., MSB or Entropy-based) and the size of the hash key. The rules distribution provides sufficient statistics regarding the location of each rule in each sub-table, and therefore, assuming uniform packets distribution, one can evaluate in advance the required time for execution of the linear search in each table. The proposed entropy-based hashing makes use of the given rule set data structure and leads to uniformly distribution of the rules. The experimental results demonstrate the superiority of the entropy-based hashing using 18-bits key length (for both ACL and FW).

Consider a large rules-set R with N_R rules, where $R = \{r_1, r_2, \ldots, r_{N_R}\}$, and each rule r_i is represented by a sequence of n bits, $B = \langle b_{i1}, b_{i2}, \ldots, b_{in}\rangle$. We propose to use information entropy-based criterion - the maximal Information Gain (IG), to find a uniform hash key with $k < n$ bits. A proper hash key should yield a relatively small number of rules replications, considering the presence of "*wildcard*" bits in the rule's fields. The IG reflects the decrease in mutual entropy after choosing a specific bit to be included in the hash key, and therefore may be considered as a good criterion for selecting the appropriate bits.

The entropy of a given ruleset R is given by:

$$H(R) = -\sum_{r_i \in R} P(r_i) \log_2 P(r_i) \tag{1}$$

where $P(r_i)$ is the probability of appearence of rule r_i.

The selection of a proper k bits key should consider both the distribution of the binary values '0' and '1' and the number of wildcards in each rule. Consider that each rule r_i contains $m_i^{\emptyset}$ *wildcards*, therefore the number of distinct combinations of B associated with rule r_i is $2^{m_i^{\emptyset}}$, and for N_R rules there are $N_{tot} = \sum_{i=1}^{N_R} 2^{m_i^{\emptyset}}$ combinations representing R (i.e. the number of rules considering their replications). The probability $P(r_i)$ can be expressed in terms of $m_i^{\emptyset}$ as:

$$P(r_i) = 2^{m_i^{\emptyset}} / N_{tot} \tag{2}$$

Therefore, the entropy of the ruleset R given in Eq. (1) can be expressed as:

$$H(R) = \frac{1}{N_{tot}} \sum_{i=1}^{N_R} 2^{m_i^{\emptyset}} \left(\log_2 N_{tot} - m_i^{\emptyset} \right) \tag{3}$$

The IG for a given rule-set R and bit b_j is given by the following equation:

$$IG(R, b_j) = H(R) - H(R|b_j) = -\sum_{i=1}^{N_R} P(r_i)\log_2(r_i) - \sum_{i=1}^{N_R} P(r_i|b_j)\log_2\left(P(r_i|b_j)\right) \quad (4)$$

where $P(r_i|b_j)$ is the probability of rule r_i given bit b_j.

Considering the rules replications, Eq. (4) can be expressed as:

$$IG(R, b_j) = \log_2(N_{tot}) - \frac{1}{N_{tot}}\left[N_0^* \log_2 N_0^* + N_1^* \log_2 N_1^* + 2N_\emptyset\right] \quad (5)$$

where $N_0^* = N_0 + N_\emptyset$, $N_1^* = N_1 + N_\emptyset$, and $N_0, N_1, N_\emptyset$ are the total number of the replicated rules given by:

$$N_0 = \sum_{\{r_i|b_j=0\}} 2^{m_i^\emptyset},\ N_1 = \sum_{\{r_i|b_j=1\}} 2^{m_i^\emptyset},\ N_\emptyset = \sum_{\{r_i|b_j=\emptyset\}} 2^{m_i^\emptyset - 1}$$

Consider a rule matrix composed of N_R rows and n columns. Each row represents a specific rule, composed of a sequence of n bits, $\langle b_{i1}, b_{i2}, \ldots, b_{in}\rangle$. Equation (4) is used to calculate the IG for each column in the rule matrix (column j is composed of the b_{ij} bits of all rules, where $i = 1, \ldots, N_R$). The index of the column having the highest IG is used to select the first bit of the hash key (out of the n bits representing a rule). Then by applying an iterative process the additional bits having the highest IG can be found using a decision tree approach. The well-known ID3 algorithm [39] may be used to calculate such decision trees. However, using a decision tree scheme is inappropriate for extracting the proper hash key bits since different bits may be selected traversing different paths of the tree resulting in multiple keys. Moreover, search paths may be of different lengths and thus yielding hash keys of different length. Therefore, the k-bits hash key are chosen as the indexes of the columns in the rule matrix having the k highest IG values.

A heuristics approach has been adopted in choosing the specific packet header fields used for hashing. A combination of the port and protocol fields, with some selected bits from the IP source and destination addresses, were examined. Specifically, two hash keys based on maximum IG and the MSB pattern of the IP address have been examined [16]. The MSB pattern might be considered as a good choice since the wildcards are usually applied to the LSB pattern of the packet header fields. The IG entropy-based key has been used since entropy is maximized when all entries have the same probability of occurrence, and therefore a maximum entropy-based hash key tends to uniformly distribute the rules in the sub-tables [16]. Moreover, the maximum IG-based key also tends to favor bits associated with a smaller number of wildcards, resulting in a smaller number of replicated rules.

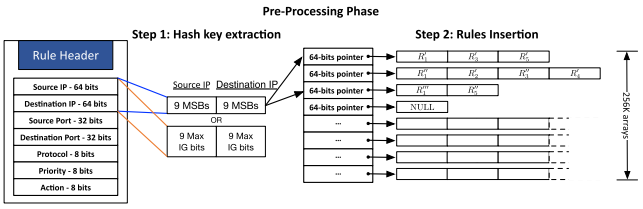

FIGURE 1. The preprocessing phase using 18 bits hash key.

The IG is calculated using the original rules-set with regards to the specific IP source and destination fields (32 bits each), forming a rule matrix of $N_R \times 64$ bits. The IG calculation results in 64 different IG values $IG(R, b_j)$, $j = 1, \ldots, 64$. For each IP address field, the highest nine IG values are selected, pointing at specific 18-bit positions within the SA and DA fields. These specific 18 bits, which represent the maximum IG are concatenated and used as the entropy-based hash key. Therefore, a hash table of size of 256k is created.

The *MSB pattern* hash key is derived by concatenating the nine *MSB bits* of each IP header field, providing 18-bits MSB hash key. IP fields may contain wildcards bits, and therefore the number of the actual rules is expanded and can be extremely large. However, since the *MSB pattern* of an IP address usually contains much fewer wildcards, it is expected to require less memory space.

The required memory space is determined by the size of the hash table and the total number of rules in the final sub-tables. Our proposed one-level entropy-based hashing using the IP fields require only 2MB for storing the entire rules database. The space complexity of the hash table is $O(2^{\sum k_i} \cdot p)$ where k_i is the size of the hash key at level i and p is the size of the pointer [16]. Therefore, the proposed approach demonstrates a complexity of $O(2^k \cdot p)$, where k is the size of the hash key and p is the size of the pointer, making it feasible for GPU implementation.

D. PRE-PROCESSING PHASE

The proposed algorithm consists of two main phases: an off-line pre-processing phase, and a real-time classification phase. During the pre-processing phase, the partitioning of the original rules-set into smaller sub rulesets is carried out, based on the selected classification fields. A dedicated hash table is generated, and the rules are inserted according to the selected hash key into the different sub-tables. A hash table with 256k entries is created for an 18-bits hash-key, each pointing at different *sub-rules set*. Fig. 1 depicts the proposed hashing algorithm and the pre-processing phase. The 18-bits hash key is extracted from the SA and DA fields (9-bits out of 32 each) using *IG criteria* or *MSB pattern*, and the original rule-set is partitioned into 256k smaller independent sub-tables. In case an incoming packet matches more than one rule the rule with the higher priority is applied. Therefore, for efficient linear search, the rules list in each sub-table is organized in descending priority order.

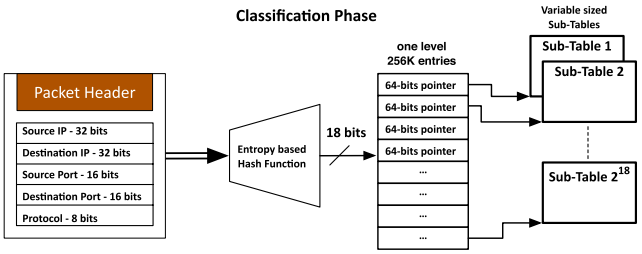

FIGURE 2. The packet classification phase.

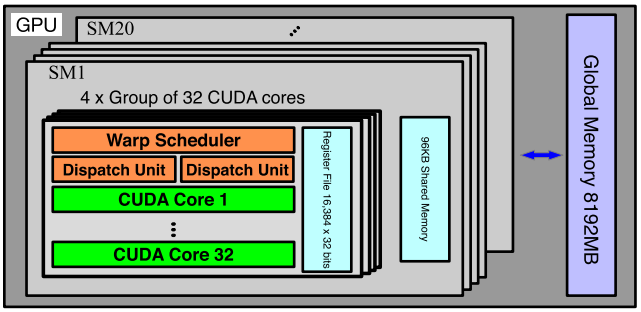

FIGURE 3. A simplified architecture of a single SM.

E. CLASSIFICATION PHASE

The classification phase is carried out in real-time and should face the wire speed requirements. The same hash key, used for partitioning the original rules-set and constructing the hash tables, is also used by the classifier. This minimizes the classifier searches to the relevant sub ruleset for which the incoming packet belongs to. Fig. 2 demonstrates the classification phase for the entropy-based hashing. First, the same 18-bits representing the maximum IG are selected from the two 32-bits IP address fields of the incoming packet header. The selected key points to a specific entry in the hash table. Then, a 64-bits pointer, stored in the specified entry, points to the address of one of the 256k sub-tables, which contains a sub-list of possible matching rules. Finally, a simple linear search is applied to the selected sub-table. Since the rules are organized in descending priority order, only the first rule which matches the incoming packet is considered (i.e., the matching rule with the highest priority), and the packet is classified and assigned the appropriate flow and action accordingly. In case no match is found, a NULL is returned to define a default action for the incoming package. Figure 2 presents the classification process and possible rules distribution.

III. PARALLEL IMPLEMENTATION USING GPU

A. GPU ARCHITECTURE

This section reviews the architecture and the programming model of the NVIDIA GTX-1080 GPU [40], [41]. The main challenge is is to efficiently utilize the GPU parallelization ability for the specific packet classification problem. GPUs have been designed to support data-parallelization applications, for which many scalar processing units can process different parts of the data, in parallel. This process is carried out using thousands of threads in parallel, where all threads execute the same *kernel program*. A set of concurrently executing threads is called a *thread block*, and at run-time, each *block* is assigned to a specific *Streaming Multiprocessor* (SM), each contains multiple *Stream Processors* (SPs). Threads that belong to the same block share a fast-local memory and a large set of registers within each SM. The SM splits the blocks into warps which consist of a fixed number of threads and is the basic execution unit on GPU. Following, the *Single Instruction Multiple Threads* (SIMT) parallel programming model, all the SPs execute the same instruction in lockstep for a *warp* (typically a set of 32 threads). A warp scheduler is responsible for selecting some independent warps at a time and issues one instruction for each warp. A global DDR memory is typically used for copying the data from and to the host CPU memory. The GPU programming model efficiency fits the given packet classification problem, for which each incoming packet is examined for a match against multiple classification rules in parallel. Figure 3 shows a simplified architecture scheme of a single SM of the NVIDIA GTX 1080 GPU [40], [41]. The selected GPU consists of 20 SMs, each containing 128 SPs running at 1607MHz. The on-chip memory contains: 256kB register file, 48kB L1 cache, and 96kB shared memory unit. The off-chip memory includes 8GB device memory and 2048kB L2 cache which is shared among all SMs. Registers which are allocated to a specific thread cannot be shared with other threads. The SM quad warp scheduler allows up to four warps to be executed concurrently. The two dispatch units enable dispatching of two independent instructions per warp in each cycle and thus hiding latency. The GTX-1080 supports up to 32 blocks, 64 warps and 2048 threads per SM. Therefore, a maximum of 40960 threads can be executed concurrently. The chosen GTX-1080 GPU is one of the commonly used NVIDIA graphic processors. The proposed approach can be implemented on any other GPU platform provided that its global memory is large enough to contain the rule-sets including replication. A selection of a specific GPU may affect the performance (due to different clock frequency, cache and memory sizes, no. of SMs, no. of warps, number of threads, etc.). For a fair comparison with other related work, the proposed approach has been implemented also on the GTX-580.

Two common metrics are considered to evaluate GPU utilization: *Occupancy* and *Load efficiency* [22], [41]. Occupancy relates to the number of active wraps running concurrently, while Load Efficiency expresses the amount of coalesced memory accesses (addressing the same cache line).

The following factors limit the number of blocks which can execute concurrently per SM. The number of warps and the number of blocks per SM, as well as the local resources (registers and shared memory) per SM. The number of active warps should be equal to the maximum supported active warps, to achieve 100% occupancy. Therefore, for the

target GTX1080 GPU platform, which supports a maximum of 64 active warps per SM and 32 active blocks, we have assigned 64 threads per block (two warps per block), resulting in actually 64 active warps. Each SM can support up to 32 thread blocks (resulting in a maximum of 640 blocks per kernel), and each block consists of 64 threads, and is responsible for classifying a packet against 64 rules, i.e., assigning one thread per rule. The occupancy can be increased by increasing the block size, i.e., increasing the number of warps per block. Since each SM has a set of registers and a fixed amount of shared memory shared by all active threads, reducing the number of registers and memory needed per thread can also increase occupancy. Following the above guideline, we have tried to maintain as many active warps and threads as possible throughout the execution of the kernel, while also having a more balanced workload among the warps in each block. A careful analysis has been done to keep the local resource usage low enough to support multiple active thread blocks per SM.

The *Load efficiency* measures the amount of *coalesced* memory accesses and is defined as the ratio between the number of requested bytes and the total number of actually fetched bytes. To minimize DDR bandwidth, the GPU should access global memory using as few transactions as possible. Therefore, a special effort has been made to minimize the number of transfers to/from the CPU using one large transfer rather than many small ones. Load efficiency can be increased by accessing consecutive memory blocks while fetching data from the device memory into L1 cache, shared memory, and registers. While the input packets are copied to the shared memory, the partial set of rules is copied directly to the registers of the specific thread. Therefore, all match comparisons are carried out using a low latency memory. Although *CUDA kernels* are executed using a large number of blocks, the required resources per-block should be limited to allow more blocks to run in parallel [41].

B. PARALLEL IMPLEMENTATION OF THE PROPOSED ALGORITHM

This section introduces the parallel implementation of the proposed packet classification algorithm exploiting the GPU GTX-1080 parallelism. An iterative algorithm for parallel packet classification using two levels of parallelism is suggested. In each iteration, an incoming packet is matched against 64 different rules in parallel by assigning 64 threads per block (per SM). Moreover, up to 20 different packets are classified in parallel utilizing the total available SMs. Each thread is responsible for examining the incoming header packet (5 tuples) against one rule at a time and performs nine basic comparison operations (two range comparisons for each of the SA, DA, SP, DP fields, and one exact match for the Protocol field). At the end of each iteration, each thread produces a local rule match decision. In the case of more than one rule match, the action related to the rule with the higher priority is carried out. Otherwise, in case of no match, the 64 threads are re-assigned to examine an additional batch of 64 rules. This process continues until all the rules in the specific sub-table are examined. For a sub-rule-set of N_s rules, each thread is responsible for examining the incoming packet against a maximum of $N_s/64$ rules. A *default action* (i.e., ignore or denied) is defined in case no match is found.

Since each SM can handle a different incoming packet simultaneously, the proposed algorithm allows the examination of up to 20 incoming packets in parallel, comparing each packet header against 64 rules. This means that 1280 (64×20) rules are examined in a basic time unit. This approach has a significant advantage in cases concurrent classification of multiple packet streams is required. Also, since each incoming packet is processed by a specific SM, that is responsible for checking the packet against the whole rule-set, there is no need for sharing intermediate classification results between SMs through the global memory. Moreover, since there is no need for synchronization between SMs, as soon as a specific SM completes a packet classification, it is immediately assigned with a new packet.

An alternative parallelism approach suggests partitioning the rule-set among the available SMs (or thread blocks), such that each SM is responsible for checking a batch of incoming packets only against part of the rule-set [22]. This means that classifying a packet against all the rules requires passing each packet among all SMs. Varvello *et al.* [22] suggest concurrently matching of a given rule against 64 packets using 64 threads. This approach requires an additional final classification phase for merging all intermediate results obtained for the partial rule-set in each SM. This alternative is not appropriate for large rule-sets where repeated rule partitioning among the SMs is required. Moreover, Varvello's approach results in a more complex merging decision and needs several kernels activations to complete the classification for each packet. The performances of our proposed approach and Varvello's approach seem to be similar for the worst-case scenario. However, our proposed approach promises faster classification since it utilizes the fact that the rule-set is pre-ordered by priority. Therefore the classification process can be terminated as soon as a match is found.

Let's assume a priority-ordered rule-set with N_R rules and define the time unit required for matching one packet against a given rule as t_b. The time required for matching N_P packets against a sub-table that contains N_S rules utilizing 20 SMs is given by:

$$T = (N_P/20) \cdot \lceil N_S/64 \rceil \cdot t_b \tag{6}$$

The proposed algorithm suggests distributing the rule-set into 256k sub-tables (for 18-bits hash key), while the matching process is based on batches of 64 rules (utilizing 2 warps in each SM). The sub-table containing the maximum number of rules defines the number of batches needed to complete the classification and therefore determines the worst-case scenario. For example, the largest sub-table for a rule-set of 100k contains up to 160 rules, and therefore up to three batches, $\lceil N_S/64 \rceil = 3$, are needed in the worst case. Hence, the time required to classify a single packet,

assuming a 1607MHz clock, is 16.8ns (9 ALU operations × 0.622ns × 3). Therefore, the proposed algorithm can easily face a wire speed up of 10Gbps for which the incoming packet inter-arrival time (in the worst case for 64 bytes packets) is about 51.2 nanoseconds. Exploiting the GPU parallelism utilizing all 20 SMs, the time required to classify 1M packets is 840μs (50k packets per SM × 16.8ns). The total kernel time should also consider the extraction of the hash key and the memory accesses required for fetching the rules into the register file of each thread.

The proposed hashing approach guarantees that all the rules associated with a specific packet are placed in the same sub-table. Each SM executes an instance of the same GPU kernel and is responsible for matching a packet against 64 different rules concurrently. The classification is carried out iteratively by fetching batches of 64 rules from a specific sub-table until a match is found. All 64 threads carry out the same basic operation, i.e., nine compare operations checking one rule per thread. In case the packet matches more than one rule, the action associated with the highest priority rule is applied. A packet header contains 13 relevant bytes (using five-tuple), while a typical rule may contain up to 29 bytes (two range bounds for each IP addresses and ports, protocol, action type (accept or deny) and priority field (32 bits). A zero padding is applied to each rule data structure allowing memory data alignment purpose.

The following input variables are used by the kernel: (a) an an array **P** which includes the input packets (5-tuples fields for each packet), (b) an array **R** that contains the rules subsets organized in an ordered priority, and (c) an ***Offset*** array which contains pointers to the different sub-tables. The hash keys are derived from the 5-tuples and stored in a dedicated array **K**. The packets array **P** and rules array **R** resides in the GPU device global memory. To maximize parallelism, the packets array **P** is partitioned among several blocks, such that each block is responsible for checking one packet against its corresponding rules sub-tables. A pool of 100k packets is defined by setting the number of blocks to 100k, while the size of each block is set to 64 threads to allow matching of a packet against 64 rules in parallel. The inspected packets are fetched from the global memory to the specific SM shared memory, while the associated rules are copied from the global memory to the SM thread registers in batches of 64 rules. Each thread is responsible for inspecting one packet against a given rule performing nine compare operations. The matching results are temporarily stored in the SM shared memory. Whenever a match is found, the current thread block is terminated and its resources are assigned to another thread block start working on a new packet.

A further improvement is achieved by accelerating the nine comparison operations, required for testing a rule match, by using additional parallelism level. Instead of allocating one thread per rule and allowing parallel inspection of 64 rules, we have assigned eight threads per rule, thus allowing testing only 8 rules simultaneously. This approach efficiently exploits the available resources of 64 threads per SM in all the cases for which the size of the sub-table is not a multiple of 64.

Inspecting a packet against up to 64 rules, while the given resources are 64 threads, can be carried out using two approaches: (a) allocating one threads per rule allowing testing up to 64 rules simultaneously in t seconds (b) allocating eights threads per rule allowing testing up to 8 rules in $t/8$ seconds. The disadvantage of the first approach is that it requires the same time (t sec) for processing a single rule or up to 64 rules. Therefore, in case the batch size is less than 64, the second approach is preferable. For example, testing 32 rules takes t sec using the first approach and only $4\,t/8$ sec using the second approach. The required processing time is given in Eq. (7) below.

$$T = \left\lceil \frac{r}{8} \right\rceil \cdot \frac{t}{8} \tag{7}$$

The pseudo-code for the proposed linear search kernel algorithm is depicted in Algorithm.1. In the initialization phase, the shared memory is allocated to store the following variables: the packet under test, the hash key, and the matching results. Lines 3-6 copy the inspected packet fields from the global to the shared memory. Lines 8-12 extract the hash key and calculate the sub-table size. Lines 17-27 scan a batch of rules and perform nine compares per rule (line 19). Finally, in line 30, the index of the matched rule (one rule per packet) is stored in the global memory.

IV. EXPERIMENTS AND RESULTS

This section presents various simulations of the proposed algorithm using different sizes of rule-sets and different hash keys. The rules-sets are derived for two widely used applications, ACL and FW, using the common *ClassBench* toolset. For each type of application, different rule-sets are extracted, ranging from 10k up to 500k rules. The packet traces are synthesized from appropriate rule-base generating two million packets for testing the algorithm performance. The classification results are demonstrated for both CPU and GPU platforms, using 3.5GHz Intel Xeon E5-1620 v4 CPU with 32GB DDR3, and NVIDIA GTX 1080 GPU with 8GB GDDR5X [40]. The performance evaluation of the proposed algorithm is measured using two criteria: average/worst memory accesses and classification time. Since a linear search is carried out to find a match against the rules list in a sub-table, the algorithm complexity linearly depends on the number of rules in the tested sub-table. The number of memory accesses required for matching a given packet is proportional to the number of rules in the appropriate sub-table. Therefore, the longest sub-table determines the worst-case scenario. Moreover, in the case of a burst of packets belonging to the same flow, the performance of the linear search algorithm may significantly decrease. Especially if the matching rule is located at the end of a large sub-table and repeatedly addressed by a long burst of packets. The classification performance is given in terms of throughput (number of packets per second). Performance comparison

Algorithm 1 Parallel Linear Search Algorithm

Inputs: An array of all rules R //Global
Array of packets P //Global
Hash key mask K_{mask} //Shared
Offsets array (pointers to sub-tables) O_{ptr} //Global
Outputs: Array of match results A_{Res} //one entry per packet

Initialization:
The number of threads is assigned to 64 ($blockDim \leftarrow 64$)
CUDA assigns a block of 64 threads to each packet
Allocate shared memory for: packet's fields -P_{field}, hash key -K_{hash}, size of the rules sub-table - ST_{size}, match results vector -RES[64], and a common match flag -$Block_{match}$

1: $Tid \leftarrow threadIdx$
2: $Packet_{idx} \leftarrow blockIdx \times Packet_{size}$ //$Packet_{size} = 20$ bytes
// Threads 0-4 manage copying the packet to shared mem.
3: **if** $Tid < 5$ **then** //5 tuples
4: $field\_offset \leftarrow Tid \times 4$ // 4 bytes for each field
5: $P_{field}[field\_offset] \leftarrow P[Packet_{idx} + field\_offset]$
6: **end if**
7: syncthreads()
// Thread 0 extracts the hash key
8: **if** $Tid = 0$ **then**
9: $K_{hash} \leftarrow$ keyExtract(P_{field}, K_{mask}) //extract the hash key
10: $ST_{offset} \leftarrow O_{ptr}[K_{hash}]$ //sub-table offset
11: $ST_{size} \leftarrow O_{ptr}[K_{hash} + 1] - ST_{offset}$ // calculate sub-table size
12: **end if**
13: syncthreads()
// Parallel linear search
14: $Block_{size} \leftarrow 64$
15: $Block_{offset} \leftarrow 0$
16: $Block_{match} \leftarrow False$
17: **while** $(Block_{offset} + Tid < ST_{size}) \wedge (Block_{match} = False)$ **do**
18: $Thread\_Rule \leftarrow R[ST_{offset} + Block_{offset} + Tid]$
// Copy rules from global memory to register file
19: $match\_flag \leftarrow match(P_{field}, Thread\_Rule)$
20: $RES[Tid] \leftarrow match\_flag$ // update the matched rule index
21: syncthreads()
22: scan RES for highest priority matched rule //parallel reduce
23: store the index of the highest priority matched rule in RES[0]
24: update the $Block_{match}$ // true if match
25: syncthreads()
26: $Block_{offset} \leftarrow Block_{offset} + Block_{size}$ // block size is 64
27: **end while**
28: syncthreads()
29: **if** $(Tid = 0) \wedge (Block_{match} = True)$ **then**
30: $A_{Res}[blockIdx] \leftarrow RES[0]$ //one matched rule per packet
31: **end if**

against previous works shows the superiority of the proposed algorithm in terms of both classification speed and memory utilization.

A. RULE-BASE DESCRIPTION

The rule-bases are generated using real rule-set [37], a publicly available tool for benchmarking packet classification algorithms. The *ClassBench* tool uses various kernels representing common internet and transport protocols such as TCP, UDP, and ICMP. We use two of the *ClassBench* rules set: Access control list (ACL) and Firewall (FW). For each application (ACL and FW), various sets of rules are generated:

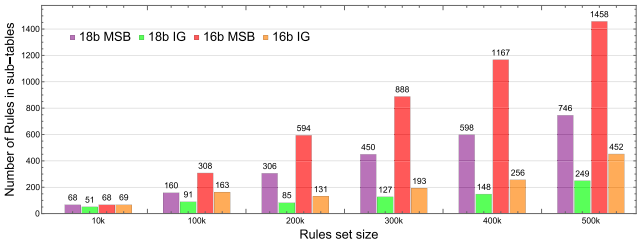

FIGURE 4. ACL rules distribution (max values).

10k, 100k, 200k, 300k, 400k, and 500k, with appropriate two million packets for each rule-set. The packet sets are correlated to the rule-sets and generated such that (a) the packets are different, and (b) each rule matches the same number of packets. The proposed algorithm uses hashing and enables partitioning a very large rules-set into smaller distributed sub-rules look-up tables based on IG and MSB pattern hash keys. For the 18-bit hash key, the rules are distributed among 256k different sub-tables with no more than about 170 and 750 rules in each table for the 100k and 500k rule-set, respectively. Since the rules may contain several wild-cards, they should be accordingly duplicated among the 256k sub-tables. Therefore, the total rules-list is significantly expended and can reach up to about 15 million rules for a 100k rule-set.

B. CPU RESULTS

This section presents the results of the proposed classification algorithm using the Intel Xeon CPU implementation. The results for both ACL and FW applications are presented using MSB and IG entropy hashing with 16 and 18 bits key length. Fig. 4 depicts the rule-sets distribution among the different sub-tables using the two hash key types with 18- and 16-bits key length for ACL. Both average and maximum sub-table sizes are presented for different sizes of rule-sets ranging from 10k up to 500k.

The MSB key typically contains less *wildcards* than the entropy key, and therefore, the size of the expanded rule-set for IG is up to 10 times greater comparing to MSB. Hence, the average sub-table size for MSB (6 rules for 100k) is also much lower than the IG (70 rules for 100k). However, the maximum sub-table size for MSB ($\sim$ 600 rules for 400k) is much larger comparing to IG ($\sim$ 150 rules for 400k). The use of 18-bits key is preferable since the size of the sub-tables is lower both for the average and maximum cases.

Fig. 5 and Fig. 6 demonstrate the throughput, and the average memory accesses for ACL. The IG is superior to the MSB key since the entropy-based key requires on average much fewer memory accesses, for rule-set sizes of 100k and more (11 vs. 264 accesses for 500k using 18-bits). The IG with 18-bits key demonstrates the best performance in terms of throughput and memory accesses for large rule-sets. For 100k rules, a throughput of 4 Mpps is achieved using entropy while the MSB demonstrates only 2 Mpps.

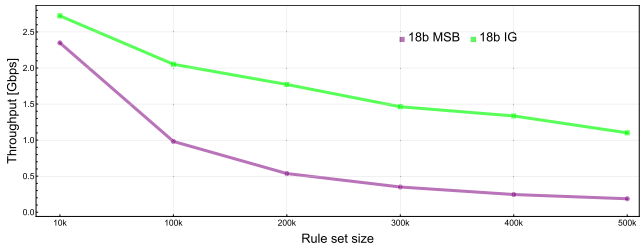

FIGURE 5. CPU ACL throughput.

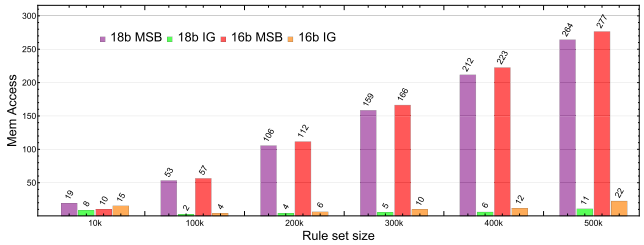

FIGURE 6. ACL average memory accesses.

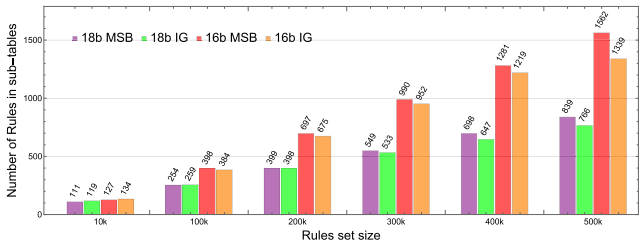

FIGURE 7. FW rules distribution (max values).

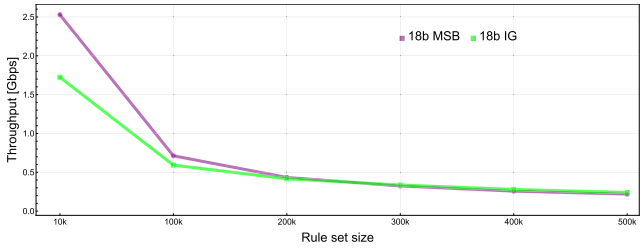

FIGURE 8. CPU FW throughput.

Fig. 7 depicts the rule-sets distribution for FW. The 18-bits key is preferable since the average sub-table size is lower (about half the size of 16-bits for large rule-sets), and therefore the linear search time is twice faster.

Fig. 8 and Fig. 9 show the throughput and average memory accesses for FW. Both hash keys demonstrate similar performance in terms of throughput and are inferior to ACL (for example, 1.2 Mpps vs. 4 Mpps for IG with 100k rules). Fig. 9 shows the average memory accesses. The 18-bit key requires much less memory accesses (eg. 54 vs. 106 memory accesses for 100k rules, using entropy). We conclude that the entropy-based key is a better choice for ACL and promises higher throughput compared to MSB. However, the throughput using entropy decrease from 5.3Mpps for 10k down to 2.1Mpps for 500k rules.

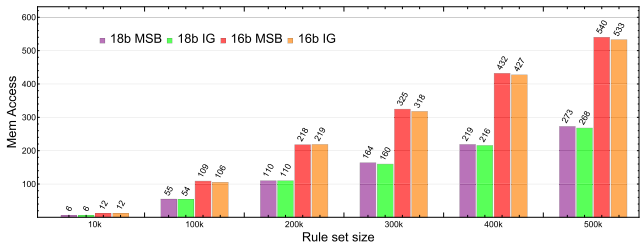

FIGURE 9. FW average memory accesses.

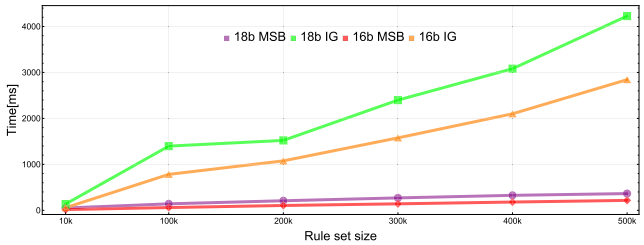

FIGURE 10. ACL pre-processing time.

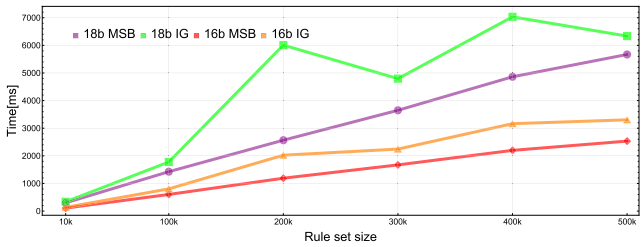

FIGURE 11. FW pre-processing time.

Fig. 10 and Fig. 11 depict the required pre-processing time for the MSB and IG as a function of the rule-sets sizes for ACL and FW. As expected, the time required for distributing the extended rules-list among the sub-tables for the entropy-based key is up to 10 times higher compared to MSB for ACL, and up to twice for FW.

C. GPU RESULTS

This section presents the results of the proposed classification algorithm carried out on the GPU platform. Results for both ACL and FW applications are presented using MSB and Entropy hashing with 18 bits key length. The speedup achieved by the parallel GPU implementation is demonstrated compared to the CPU. The time needed for extracting the hashing key is negligible compared to the classification processing time and does not affect the measured thought. The GPU throughput is mainly affected by the number of rules in a sub-table and the number of threads assigned to a block (SM), which defines the processing batch size. The time needed to complete the linear search depends on the number of batches needed until a match decision can be taken.

Figure 12 shows the throughput in terms of Gbps for both ACL and FW (for 18-bits hash keys). The OC-192 standard,

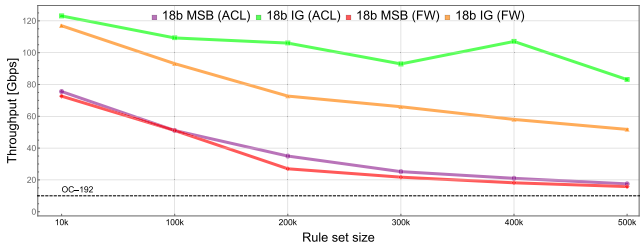

FIGURE 12. GPU throughput for ACL and FW.

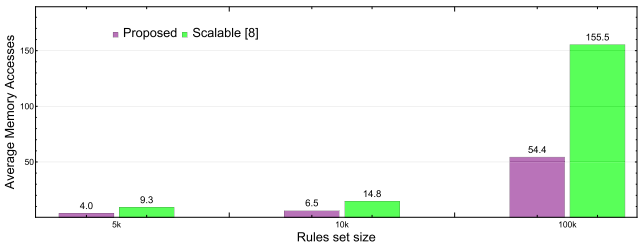

FIGURE 13. Average memory accesses.

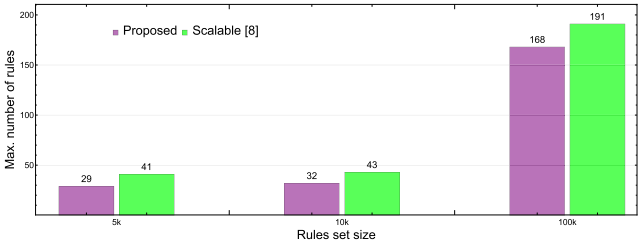

FIGURE 14. FW rule-set distribution.

demonstrating a wire-speed of 10 Gbps, is chosen as a reference throughput (horizontal dashed line), for which 64-bytes packet length is considered as the worst-case scenario.

The IG outperforms the MSB demonstrating throughput of 82 Gbps compared to 18Gbps for ACL with 500k rules. This result is expected due to the larger size of the sub-tables for the MSB key. For both keys, the target of 10Gbps wire-speed is achieved. Although the size of the sub-tables are similar for both keys in the case of FW, the IG still outperforms the MSB, demonstrating a throughput of 52Gbps compared to 16 Gbps with 500k rules. This can be explained since for IG most of the linear searches are completed within the first batch.

Fig. 12 depicts a significant speed-up achieved by the GPU compared to a single-core CPU for both ACL and FW. For ACL, a speed-up of up to 90 and 80 is demonstrated for MSB and IG, correspondingly. For FW a speed-up of up to 70 and 200 is demonstrated for MSB and IG, correspondingly. This indicates the effective implementation of the proposed algorithm exploiting the parallel capabilities of the GPU platform, utilizing the full occupancy of the GPU warps and efficient memory utilization.

D. COMPARISON WITH RELATED WORK

This section compares the performance of the proposed approach with similar previously published works. First, we evaluate the proposed one-level IG entropy-based hashing approach, compared to the two-level hashing *scalable* algorithm presented by Choi *et al.* [16]. Then, the proposed GPU-based implementation is compared to both Varvello *et al.* [22] and Zhou *et al.* [34], in terms of throughput and the number of memory access.

1) CPU-BASED CLASSIFICATION COMPARISON

For a fair comparison with the work done by Choi *et al.* [16] we have fully implemented the scalable algorithm presented in [16], using the same rules-sets and the same CPU platform for performance evaluation. Both algorithms have been tested using a 3.5GHz Intel Xeon CPU, with various rules-sets of sizes of 5k, 10k, and 100k derived from the *ClassBench*. Our proposed algorithm outperforms the packet classification presented in [16] in terms of both memory consumption and throughput.

The proposed IG-based one-level hashing approach suggests much better memory utilization. While [16] presents two-level hashing (first level hashing based on port and protocol fields, and then a second level hashing based on IP) we suggest a one-level hashing approach using mutual information gain (IG) instead of the maximum entropy proposed by [16].

According to [16], memory consumption of 32GB is required for the two-level hashing data structure, not including the expanded rule-list. For 500k rules-set, the number of the extended rules is about 200M rules, requiring an additional 6.4GB memory space (32 bytes per rule). This large memory requirement is not acceptable for real-time implementation using a commercial GPU with a typical memory size of 8-12GB. Using one-level hashing with 18-bits IG entropy-based hash key requires a total of 2MB for the main hash table (which contains pointers to sub-tables). For 100k the expanded FW rules-list contains ∼32M rules, and therefore a total memory size of ∼256MB is needed.

Fig. 13 shows the average memory accesses for the two algorithms, for the FW scenario. For a fair comparison, we choose the hash key demonstrating the best performance in [16], i.e., the maximum entropy pattern. The scalable algorithm demonstrates much more memory accesses on average (a factor of about 2-3), for all rule-sets, compared to our algorithm. Therefore, the proposed IG entropy-based algorithm outperforms the scalable algorithm in terms of classification runtime. Fig. 14 depicts the FW rules-set distribution among the various sub-tables for the two algorithms. The maximum sub-table size for the 100k rules-list is slightly lower for the proposed IG algorithm compared to the scalable algorithm (168 vs. 191 rules). Therefore it demonstrates better performance also for the worst-case linear search scenario.

To extend the CPU-based comparison we compare the performance of the proposed approach also with the serial implementation of the multilayer approach presented in [22].

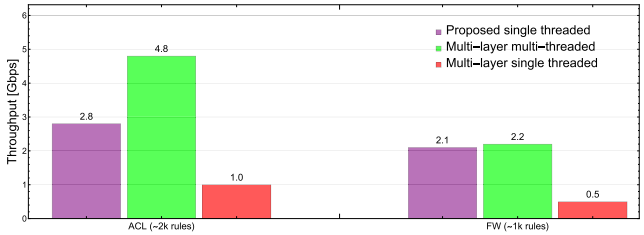

FIGURE 15. CPU throughput comparison (realistic rules).

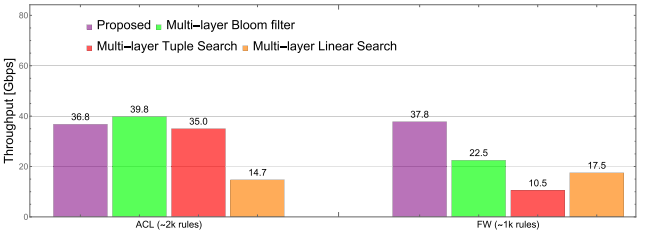

FIGURE 16. GPU throughput comparison (realistic rules).

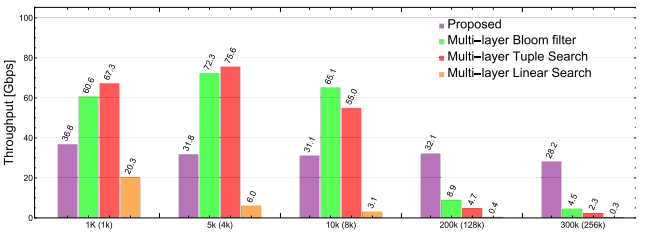

FIGURE 17. GPU throughput comparison using realistic rule-sets (proposed alg.) and synthetic rule-sets for the others.

The performance evaluation is carried out for both ACL and FW applications using the same CPU platform. Fig. 15 depicts the throughput of the IG proposed algorithm (for a single-thread CPU), and the *multilayer algorithm* (for single-thread and 8-threads) for a realistic scenario using real rule-sets as described in [22]. The results of our IG entropy-based hashing algorithm, are compared against the *multilayer tuple search* for ACL and FW (with 2k and 1k rules, correspondingly). The proposed algorithm outperforms the multilayer single thread demonstrating 2.8 and 2.1 Gbps, comparing to 1 and 0.5 Gbps, for ACL and FW, respectively.

The reasons for the difference in throughput can be explained since the proposed entropy-based key requires, on average, much fewer memory accesses. This is also the reason for the higher throughput achieved for ACL compared to FW in both approaches. Moreover, although the proposed algorithm is carried out using a single thread CPU, it demonstrates similar throughput (2.13 Gbps) compared to the multi-thread implementation (2.2 Gbps) for FW. A speedup factor of about four is achieved while using an 8-threads CPU compared to a single thread [22]. Therefore, considering this speedup factor, also for ACL, our IG algorithm has the potential to outperform the multilayer 8-threads.

2) GPU-BASED CLASSIFICATION COMPARISON

This section compares the GPU-based implementation of the proposed approach against the *multilayer* classification algorithm and the parallelism approach presented by [22]. Varvello *et al.* suggest partitioning the rule-set among the available SMs, such that each SM is responsible for checking a batch of incoming packets only against part of the rule-set. This approach requires an additional final classification phase for merging the initial intermediate results achieved for the partial rule-set in each SM. Moreover, this approach is not appropriate for a large rule-set where repeated iterative rule partitioning among the SMs is required. This leads to more complex merging decisions and multiple *kernels* activations to complete the classification of each packet.

We propose an alternative parallelism approach suggesting that each SM is responsible for classifying a packet against the whole rules-set (64 rules in parallel). Therefore, eliminating the need for sharing intermediate classification results between the SMs through the global memory. Although in the worst-case the performances of both methods are similar, our approach promises faster classification. We utilize the fact that the rules-set is preordered by priority, and therefore the classification process is terminated as soon as a match is found in one of the active SMs. For a fair comparison with the work done in [22] the same GPU platform, the GTX 580, has been used for evaluating the proposed IG algorithm. Fig. 16 depicts the GPU throughput for the proposed IG algorithm and three other GPU-based algorithms presented in [22]: *Bloom Search*, *Tuple Search*, and *Linear Search*. The results are given for the realistic scenario described in [22] using real rule-sets derived from the *ClassBench* tool (with 2k and 1k rules for ACL and FW respectively), for minimum-sized 64-byte packets. The proposed IG algorithm outperforms the other three algorithms for FW, demonstrating a throughput of 37.8 Gbps compared to 22.5 Gbps for the *Bloom Filter*. For ACL, similar throughputs are demonstrated except for the linear search. The GPU proposed implementation demonstrates similar results for both ACL and FW since both applications are characterized by similar rules' distribution for small rule-sets (up to 10k). However, Varvello *et al.* [22] demonstrate a higher throughput for ACL compared to FW. This can be explained by the different number of classes for these applications. While ACL rules are partitioned into only 80 classes, the FW rules result in 221 classes [22].

The comparison is extended for larger rule-sets of up to 256k rules. While the evaluation conducted in [22] uses *synthetic* rule-sets with sizes of 1k, 4k, 8k, 128k, and 256k, the proposed IG algorithm is evaluated using real rule-sets derived from *ClassBench* with 1k, 4k, 10k, 200k and 300k rule-sets, correspondingly. Fig. 17 shows the throughput achieved for ACL by the proposed algorithm and by the three GPU-based algorithms presented in [22]. The superior results of [22] for small rule-sets are probably achieved due to the use of synthetic rule set, while the results for the proposed method are shown for realistic rules. Moreover, for large rule-sets, the throughput presented in [22] is dramatically decreased

due to a large number of required classes. Our proposed approach is scalable demonstrating high throughput also for large rule-sets since the matching of each incoming packet is carried out simultaneously against multiple rules (64 rules per SM).

Although, for small rule-sets (up to 10k rules) both the *Bloom* and *Tuple* algorithms show better performance, our proposed algorithm demostrates significant higher throughput for large rule-sets. For e.g., a throughput of 32.1 and 26.8 Gbps is achieved compared to 8.9 and 4.5 Gbps for the *Bloom Search*, for 200k and 300k rules, respectively. The improved throughput achieved by the proposed approach should be even higher considering the complexity of the *ClassBench* real rule-sets compared to the *synthetic rules* used in [22]. This can be confirmed by analyzing the results demonstrated in [22], for which, the throughput of 69 Gbps achieved with a synthetic rule-set drops to 39.8 Gbps while using the *ClassBench* rule-sets for the same number of rules (2k rules for ACL).

Finally, the proposed algorithm is also compared to the GPU-based packet classification presented by Zhou *et al.* [34] implementing a range-tree search in GPU. Although this algorithm scales well for a limited range of rules-set sizes (from 512 to 4k rules), the throughput dramatically drops as the rule-set increases [34]. Our proposed IG algorithm outperforms Zhou's algorithm, demonstrating 31.8 Gbps compared to 22.6 Gbps for the 4k synthetic rule-set, although they used a faster GPU (Tesla K20) platform.

V. CONCLUSION

This research presents a novel efficient parallel algorithm for packet classification supporting very large rule-sets. The proposed algorithm uses a one-level hashing data structure to enable partitioning a large rules-set into smaller uniformly distributed sub-rules look-up tables. The classification algorithm is based on entropy hashing using Information Gain (IG), and Most Significant Bit (MSB) pattern hash keys. This approach minimizes the classifier searches only to the relevant, appropriate look-up table, and therefore, the classification process is significantly shortened.

A further speed-up factor is achieved by parallelizing the classification algorithm using a Graphics Processing Unit (GPU). The parallel implementation efficiently utilizes the multithreaded architecture of the GPU, allowing matching an incoming packet simultaneously against multiple rules (64 rules per each SM), and handling many packets in parallel (a packet per SM block). Moreover, using the IG-based hashing key combined with the proposed one-level hashing dramatically decreases the memory consumption. Simulation results show that the proposed algorithm outperforms existing classifiers in terms of both speed up and memory utilization. A significant classification speed up, of about 200, is achieved by using the GPU architecture compared to other serial implementations.

The proposed GPU-based implementation is memory efficient and demonstrates high throughput with regard to the common wire-speed of OC-192 (10 Gbps). The performance of the proposed algorithm is examined using two common data communication applications: Access Control List (ACL) and Firewall (FW), using realistic rule sets generated by the *ClassBench* toolset. A throughput of 80 Gbps and 50 Gbps is achieved for ACL and FW applications, respectively, for 500k rule-sets. Unlike most published classification algorithms, the proposed algorithm can handle very large realistic rule-sets (of up to 500k and more) in real-time, facing the high throughput of today's wire speed. To the best of our knowledge, this is the first work facing such large and realistic rule-sets.

The performance of the proposed algorithm is compared against previous GPU-based packet classification algorithms [22], [34]. This work presents an alternative parallelism approach to the multilayer classification algorithm presented by Varvello *et al.* by eliminating the need for sharing intermediate classification results between the SMs through the global memory. Our proposed algorithm outperforms the three other GPU-based algorithms presented in [22], demonstrating significant higher throughput for large rule-sets. The superiority of the proposed algorithm has been demonstrated also comparing to Zhou's GPU-based algorithm [34], demonstrating better throughput.

REFERENCES

[1] J. Yuan, Z. Li, and R. Yuan, "Information entropy based clustering method for unsupervised Internet traffic classification," in *Proc. IEEE Int. Conf. Commun.*, May 2008, pp. 1588–1592.

[2] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, Sep. 2005.

[3] M. Dixit, A. Kale, M. Narote, S. Talwalkar, and B. V. Barbadekar, "Fast packet classification algorithms," *Int. J. Comput. Theory Eng.*, vol. 4, no. 6, p. 1030, 2012.

[4] X.-A. Bi, Y. Zhou, and J. Yu, "Clustering boundary cutting for packet classification based on distribution density," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. IEEE Int. Conf. Ubiquitous Comput. Commun. (ISPA/IUCC)*, Dec. 2017, pp. 661–666.

[5] W. Pak and Y.-J. Choi, "High performance and high scalable packet classification algorithm for network security systems," *IEEE Trans. Dependable Secure Comput.*, vol. 14, no. 1, pp. 37–49, 2015.

[6] Y.-C. Cheng and P.-C. Wang, "Packet classification using dynamically generated decision trees," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 582–586, Feb. 2015.

[7] Y. R. Qu and V. K. Prasanna, "High-performance and dynamically updatable packet classification engine on FPGA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 197–209, Jan. 2016.

[8] Y. Qu, S. Zhou, and V. K. Prasanna, "Scalable many-field packet classification on multi-core processors," in *Proc. 25th Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2013, pp. 33–40.

[9] R. Leira, P. Gomez, I. Gonzalez, and J. E. L. de Vergara, "Multimedia flow classification at 10 gbps using acceleration techniques on commodity hardware," in *Proc. Int. Conf. Smart Commun. Netw. Technol. (SaCoNeT)*, Jun. 2013, pp. 1–5.

[10] K. Lee and S. Yun, "Hybrid memory-efficient multimatch packet classification for NIDS," *Microprocessors Microsyst.*, vol. 39, no. 2, pp. 113–121, Mar. 2015.

[11] M. Kekely, L. Kekely, and J. Kořenek, "General memory efficient packet matching FPGA architecture for future high-speed networks," *Microprocessors Microsyst.*, vol. 73, Mar. 2020, Art. no. 102950.

[12] D.-Y. Chang and P.-C. Wang, "TCAM-based multi-match packet classification using multidimensional rule layering," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 1125–1138, Apr. 2016.

[13] I. Syafalni, T. Sasao, X. Wen, S. Holst, and K. Miyase, "Soft-error tolerant TCAMs for high-reliability packet classifications," in *Proc. IEEE Asia–Pacific Conf. Circuits Syst. (APCCAS)*, Nov. 2014, pp. 471–474.
[14] A. X. Liu, C. R. Meiners, and E. Torng, "Packet classification using binary content addressable memory," *IEEE/ACM Trans. Netw.*, vol. 24, no. 3, pp. 1295–1307, Jun. 2016.
[15] X. Dong, M. Qian, and R. Jiang, "Packet classification based on the decision tree with information entropy," *J. Supercomput.*, pp. 1–15, Jan. 2018, doi: 10.1007/s11227-017-2227-z.
[16] L. Choi, H. Kim, S. Kim, and M. H. Kim, "Scalable packet classification through rulebase partitioning using the maximum entropy hashing," *IEEE/ACM Trans. Netw.*, vol. 17, no. 6, pp. 1926–1935, Dec. 2009.
[17] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A storage-efficient, collision-free hash-based network processing architecture," *ACM SIGARCH Comput. Archit. News*, vol. 34, pp. 203–215, May 2006.
[18] H. Lim, N. Lee, G. Jin, J. Lee, Y. Choi, and C. Yim, "Boundary cutting for packet classification," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 443–456, Apr. 2014.
[19] T. Ganegedara, W. Jiang, and V. K. Prasanna, "A scalable and modular architecture for high-performance packet classification," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1135–1144, May 2014.
[20] P. Gupta and N. McKeown, "Packet classification on multiple fields," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 147–160, Oct. 1999.
[21] P.-C. Wang, "Scalable packet classification using a compound algorithm," *Int. J. Commun. Syst.*, vol. 23, nos. 6–7, pp. 841–860, 2010.
[22] M. Varvello, R. Laufer, F. Zhang, and T. V. Lakshman, "Multilayer packet classification with graphics processing units," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2728–2741, Oct. 2016.
[23] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 135–146, Oct. 1999.
[24] V. Srinivasan, "A packet classification and filter management system," in *Proc. IEEE INFOCOM . Conf. Comput. Commun. 20th Annu. Joint Conf. IEEE Comput. Commun. Soc.*, Apr. 2001, pp. 1464–1473.
[25] K. Zheng, Z. Liang, and Y. Ge, "Parallel packet classification via policy table pre-partitioning," in *Proc. GLOBECOM IEEE Global Telecommun. Conf.*, Dec. 2005, p. 6.
[26] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw. Special Issue*, vol. 15, no. 2, pp. 24–32, Mar. 2001.
[27] T. Y. C. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE INFOCOM Conf. Comput. Commun. 19th Annu. Joint Conf. IEEE Comput. Commun. Societies*, Mar. 2000, pp. 1213–1222.
[28] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. SIGCOMM*, 2003, pp. 213–224.
[29] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, Oct. 1998.
[30] X. Sun, S. K. Sahni, and Y. Q. Zhao, "Packet classification consuming small amount of memory," *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 1135–1145, Oct. 2005.
[31] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010.
[32] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 9, pp. 1668–1680, Sep. 2012.
[33] K. Kang and Y. S. Deng, "Scalable packet classification via GPU metaprogramming," in *Proc. Design, Autom. Test Eur.*, Mar. 2011, pp. 1–4.
[34] S. Zhou, S. G. Singapura, and V. K. Prasanna, "High-performance packet classification on GPU," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2014, pp. 1–6.
[35] B. Yang, X. Wang, Y. Xue, and J. Li, "DBS: A bit-level heuristic packet classification algorithm for high speed network," in *Proc. 15th Int. Conf. Parallel Distrib. Syst.*, Dec. 2009, pp. 260–267.
[36] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[37] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.
[38] Y. Rekhter and T. Li, *An Architecture for IP Address Allocation With CIDR*, document IETF Internet draft RFC 1518, Sep. 1993. [Online]. Available: http://www.isi.edu/in-notes/rfc1518.txt/
[39] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.
[40] *GeForce GTX 1080 Whitepaper*, NVIDIA, Santa Clara, CA, USA, 2016.
[41] *Cuda C Programming Guide*, NVIDIA, Santa Clara, CA, USA, 2014.

SHLOMO GREENBERG (Member, IEEE) received the B.Sc., M.Sc. (Hons.), and Ph.D. degrees in electrical and computer engineering from the Ben-Gurion University of the Negev, Beer-Sheva, Israel, in 1976, 1984, and 1997, respectively. He is currently a Staff Member with the Department of Electrical and Computer Engineering, Ben-Gurion University of the Negev. His primary research interests are computer architecture, wireless communication, image, and digital signal processing, computer vision, and VLSI low power design.

TOMER SHEPS received the B.Sc. degree from the Sami-Shamoon College and the M.Sc. degree from the Ben-Gurion University of the Negev, Beer-Sheva, Israel, in 2013 and 2018, respectively, both in electrical and computer engineering. His primary research interests are computer architecture, networking, wireless communication, digital signal processing, computer vision, and VLSI design.

DAVID A. LEON received the B.Sc. degree in communication systems engineering from the Ben-Gurion University of the Negev, Beer-Sheva, Israel, in 2017, where he is currently pursuing the master's degree with the School of Electrical and Computer Engineering. His primary research interests are parallel and concurrent programming, computer architecture, and machine learning.

YEHUDA BEN-SHIMOL (Member, IEEE) received the B.Sc., M.Sc. (Hons.), and Ph.D. (Hons.) degrees in electrical and computer engineering from the Ben-Gurion University of the Negev, Israel. He is currently a Senior Lecturer at the School of Electrical and Computer Engineering, Ben-Gurion University of the Negev. His main areas of interest are computer networks, design and analysis, and performance evaluation of communication protocols and networks.